44 | 容器：如何理解容器的实现机制？

2021-08-18 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 17:36 大小 16.12M

你好，我是 LMOS。

上节课我带你通过 KVM 技术打开了计算机虚拟化技术的大门，KVM 技术是基于内核的虚拟机，同样的 KVM 和传统的虚拟化技术一样，需要虚拟出一台完整的计算机，对于某些场景来说成本会比较高，其实还有比 KVM 更轻量化的虚拟化技术，也就是今天我们要讲的容器。

这节课我会先带你理解容器的概念，然后把它跟虚拟机作比较，之后为你讲解容器的基础架构跟基础技术，虽然这样安排有点走马观花，但这些内容都是我精选的核心知识，相会为你以后继续探索容器打下一个良好的基础。

什么是容器

容器的名词源于 container，但不得不说我们再次被翻译坑了。相比“容器”，如果翻译成“集装箱”会更加贴切。为啥这么说呢？

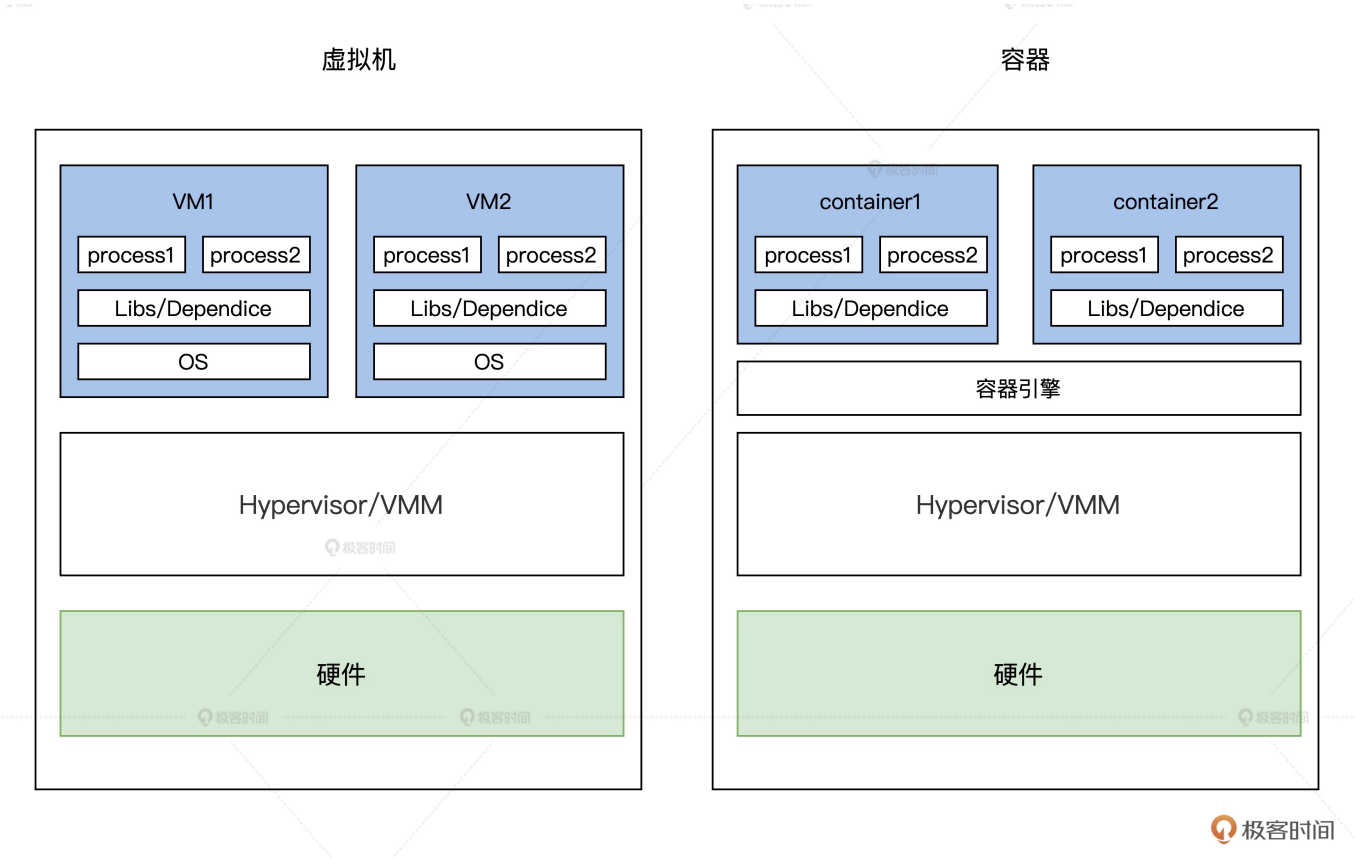
我们先从“可复用”说起，现实里我们如果有一个集装箱的模具和原材料，很容易就能批量生产出多个规格相同的集装箱。从功能角度看，集装箱可以用来打包和隔离物品。不同类型的物品放在不同的集装箱里，这样东西就不会混在一起。

而且，集装箱里的物品在运输过程中不易损坏，具体说就是不管集装箱里装了什么东西，被送到哪里，只要集装箱没破坏，再次开箱时放在里面的东西就是完好无损的。

因此，我们可以这样来理解，容器是这样一种工作模式：**轻量、拥有一个模具（镜像），既可以规模生产出多个相同集装箱（运行实例），又可以和外部环境（宿主机）隔离，最终实现对“内容”的打包隔离，方便其运输传送。**

如果把容器看作集装箱，那内部运行的进程 / 应用就应该是集装箱里的物品了，类比来看，□容器的**目的就是提供一个独立的运行环境。**

和虚拟机的对比



容器和传统虚拟机的比较

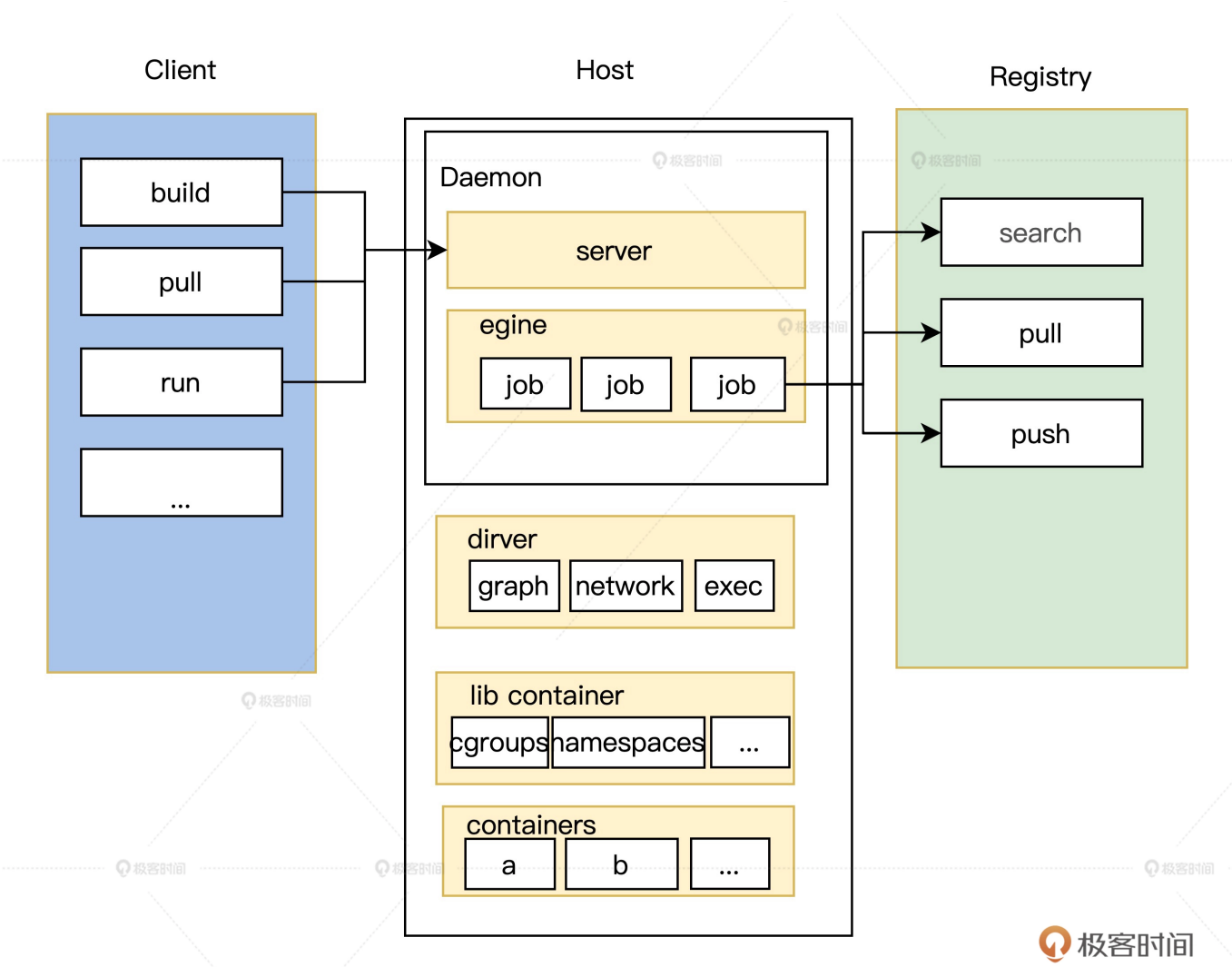
我们传统的虚拟化技术可以通过硬件模拟来实现，也可以通过操作系统软件来实现，比如上节课提到的 KVM。

为了让虚拟的应用程序达到和物理机相近的效果，我们使用了 Hypervisor/VMM（虚拟机监控器），它允许多个操作系统共享一个或多个 CPU，但是却带来了很大的开销，由于虚拟机中包括全套的 OS，调度与资源占用都非常重。

容器（container）是一种更加轻量级的操作系统虚拟化技术，它将应用程序，依赖包，库文件等运行依赖环境打包到标准化的镜像中，通过容器引擎提供进程隔离、资源可限制的运行环境，实现应用与 OS 平台及底层硬件的解耦。

为了大大降低我们的计算成本，节省物理资源，提升计算机资源的利用率，让虚拟技术更加轻量化，容器技术应运而生。那么如何实现一个容器程序呢？□我们需要先看看容器的基础架构。

看一看容器□基础架构



Docker架构示意图

容器概念的起源是哪里？其实是从 UNIX 系统的 chroot 这个系统调用开始的。

在 Linux 系统上，LXC 是第一个比较完整的容器，但是功能上还存在一定的不足，例如缺少可移植性，不够标准化。后面 Docker 的出现解决了容器标准化与可移植性问题，成为现在应用最广泛的容器技术。

Docker 是最经典，使用范围最广，最具有代表性的容器技术。所以我们就以它为例，先对容器架构进行分析，Docker 应用是一种 C/S 架构，包括 3 个核心部分。

容器客户端（Client）

首先来看 Docker 的客户端，其主要任务是接收并解析用户□的操作指令和执行参数，收集所需要的配置信息，根据相应的 Docker 命令通过 HTTP 或 REST API 等方式与 Docker daemon（守护进程）进行交互，并将处理结果返回给用户，实现 Docker 服务使用与管理。

当然我们也可以使用其他工具通过 Docker 提供的 API 与 daemon 通信。

容器镜像仓库（Registry）

Registry 就是存储容器镜像的仓库，在容器的运行过程中，Client 在接受到用户的指令后转发给 Host 下的 Daemon，它会通过网络与 Registry 进行通信，例如查询镜像（search），下载镜像（pull），推送镜像（push）等操作。

镜像仓库可以部署在公网环境，如 Docker Hub，我们也可以私有化部署到内网，通过局域网对镜像进行管理。

容器管理引擎进程（Host）

容器引擎进程是 Docker 架构的核心，包括运行 Docker Daemon（守护进程）、Image（镜像）、驱动（Driver）、Libcontainer（容器管理）等。

接下来，我们详细说说守护进程、镜像、驱动和容器管理这几个模块的运作机制 / 实现原理。

Docker Daemon 详解

首先来看 Docker Daemon 进程，它是一个常驻后台的系统进程，也是 Docker 架构中非常重要的一环。Docker Daemon 负责监听客户端请求，然后执行后续的对应逻辑，还能管理 Docker 对象（容器、镜像、网络、磁盘等）。

我们可以把 Daemon 分为三大部分，分别是 Server、Job、Engine。

Server 负责接收客户端发来的请求（由 Daemon 在后台启动 Server）。接受请求以后 Server 通过路由与分发调度找到相应的 Handler 执行请求，然后与容器镜像仓库交互（查询、拉取、推送）镜像并将结果返回给 Docker Client。

而 Engine 是 Daemon 架构中的运行引擎，同时也是 Docker 运行的核心模块。Engine 扮演了 Docker container 存储仓库的角色。Engine 执行的每一项工作，都可以拆解成多个最小动作——Job，这是 Engine 最基本的工作执行单元。

其实，Job 不光能用在 Engine 内部，Docker 内部每一步操作，都可以抽象为一个 Job。Job 负责执行各项操作时，如储存拉取的镜像，配置容器网络环境等，会使用下层的 Driver（驱动）来完成。

Docker Driver

Driver 顾名思义就是 Docker 中的驱动。设计驱动这一层依旧是解耦，将容器管理的镜像、网络和隔离执行逻辑从 Docker Daemon 的逻辑中剥离。

在 Docker Driver 的实现中，可以分为以下三类驱动。

graphdriver 负责容器镜像的管理，主要就是镜像的存储和获取，当镜像下载的时候，会将镜像持久化存储到本地的指定目录；

networkdriver 主要负责 Docker 容器网络环境的配置，如 Docker 运行时进行 IP 分配端口映射以及启动时创建网桥和虚拟网卡；

execdriver 是 Docker 的执行驱动，通过操作 Lxc 或者 libcontainer 实现资源隔离。它负责创建管理容器运行命名空间、管理分配资源和容器内部真实进程的运行；

libcontainer

上面我们提到 execdriver，通过调用 libcontainer 来完成对容器的操作，加载容器配置 container，继而创建真正的 Docker 容器。libcontainer 提供了访问内核中和容器相关的 API，负责对容器进行具体操作。

容器可以创建出一个相对隔离的环境，就容器技术本身来说，容器的核心部分是利用了我们操作系统内核的虚拟化技术，那么 libcontainer 中到底用到了哪些操作系统内核中的基础能力呢？

容器基础技术

我们经常听到，Docker 是**一个基于 Linux 操作系统下的 Namespace 和 Cgroups 和 UnionFS 的虚拟化工具**，下面我带你看一下这几个容器用到的内核中的基础能力。

Linux NameSpace

容器的一大特点就是创造出一个相对隔离的环境。在 Linux 内核中，实现各种资源隔离功能的技术就叫 Linux Namespace，它可以隔离一系列的系统资源，比如 PID（进程 ID）、UID（用户 ID）、Network 等。

看到这里，你很容易就会想到开头讲的 **chroot** 系统调用。类似于 chroot 把当前目录变成被隔离出的根目录，使得当前目录无法访问到外部的内容，Namespace 在基于 chroot 扩展升级的基础上，也可以分别将一些资源隔离起来，限制每个进程能够访问的资源。

Linux 内核提供了 7 类 Namespace，以下是不同 Namespace 的隔离资源和系统调用参数。

Namespace	隔离资源	系统调用参数	最低linux版本
PID Namespace	进程隔离	CLONE_NEWPID	2.6.24
User Namespace	UID、GID 以及根目录	CLONE_NEWUSER	3.8
UTS Namespace	主机名或域名	CLONE_NEWUTS	2.6.19
Mount NameSpace	文件挂载路径	CLONE_NEWNS	2.4.19
NET Namespace	网络栈，socket 以及网卡	CLONE_NEWNET	2.6.29
IPC Namespace	进程 IPC 通信	CLONE_NEWIPC	2.4.19
Cgroup NameSpace	Cgroup 信息	CLONE_NEWCGROUP	4.6

极客时间

1.PID Namespace：保障进程隔离，每个容器都以 PID=1 的 init 进程来启动。PID Namespace 使用了的参数 CLONE_NEWPID。类似于单独的 Linux 系统一样，每个 NameSpace 都有自己的初始化进程，PID 为 1，作为所有进程的父进程，父进程拥有很多特权。其他进程的 PID 会依次递增，子 NameSpace 的进程映射到父 NameSpace 的进程上，父 NameSpace 可以拿到全部子 NameSpace 的状态，但是每个子 NameSpace 之间是互相隔离的。

2.User Namespace：用于隔离容器中 UID、GID 以及根目录等。User Namespace 使用了 CLONE_NEWUSER 的参数，可配置映射宿主机和容器中的 UID、GID。某一个 UID 的用户，虚拟化出来一个 Namespace，在当前的 Namespace 下，用户是具有 root 权限的。但是，在宿主机上面，他还是那个用户，这样就解决了用户之间隔离的问题。

3.UTS Namespace：保障每个容器都有独立的主机名或域名。UTS Namespace 使用了的参数 CLONE_NEWUTS，用来隔离 hostname 和 NIS Domain name 两个系统标识，在 UTS Namespace 里面，每个 Namespace 允许有自己的主机名，作用就是可以让不同 namespace 中的进程看到不同的主机名。

4.Mount Namespace: 保障每个容器都有独立的目录挂载路径。Mount Namespace 使用了的参数 CLONE_NEWNS，用来隔离各个进程看到的挂载点视图，Mount Namespace 非常类似于我们前面提到的的 chroot 系统调用。

5.NET Namespace：保障每个容器有独立的网络栈、socket 和网卡设备。NET Namespace 使用了参数 CLONE_NEWNET，隔离了和网络有关的资源，如网络设备、IP 地址端口等。NET Namespace 可以让每个容器拥有自己独立的（虚拟的）网络设备，而且容器内的应用可以绑定到自己的端口，每个 Namespace 内的端口都不会互相冲突。

6.IPC Namespace：保障每个容器进程 IPC 通信隔离。IPC Namespace 使用了的参数 CLONE_NEWIPC，IPC Namespace 用来隔离 System V IPC 和 POSIX message queues，只有在相同 IPC 命名空间的容器进程之间才可以共享内存、信号量、消息队列通信。

7.Cgroup Namespace：保障容器容器中看到的 cgroup 视图，像宿主机一样以根形式来呈现，同时让容器内使用 cgroup 变得更安全。

上面讲了这么多类 Namespace，我们可以先从共性入手熟悉它们，7 类 Namespace 主要使用如下 3 个系统调用函数，其实也就是和进程有关的调用函数。

1.clone：创建新进程，根据传入上面的不同 NameSpace 类型，来创建不同的 NameSpace 进行隔离，同样的，对应的子进程也会被包含到这些 Namespace 中。

复制代码

```
1 int clone(int (*child_func)(void *), void *child_stac, int flags, void *arg);
2 flags就是标志用来描述你需要从父进程继承哪些资源，这里flags参数为将要创建的NameSpace类型，
```

2.unshare：将进程移出某个指定类型的 Namespace，并加入到新创建的 NameSpace 中， 容器中 NameSpace 也是通过 unshare 系统调用创建的。

复制代码

```
1 int unshare(int flags);
2 flags同上
```

3.setns：将进程加入到 Namespace 中。

复制代码

```
1 int setns(int fd, int nstype);
2 fd： 加入的NameSpace，指向/proc/[pid]/ns/目录里相应NameSpace对应的文件，
```

```
3 nstype：NameSpace类型
```

好了，刚刚我给你简单讲了 NameSpace 的作用以及不同类型的 NameSpace。这几种 Namespace 都是只为做一件事，**隔离容器的运行环境，**此外，NameSpace 是和进程息息相关的，NameSpace 将全局共享的资源划分为多组进程间共享的资源，当一个 NameSpace 下的进程全部退出，NameSpace 也会被销毁。

有了这么多的 Namespace 共同合作，我们才最终实现了容器进程运行环境的隔离。

现在隔离的问题已经解决，那么容器是怎么限制每个被隔离的容器的开销大小，保证容器间不会存在打架，互相争抢的问题呢？这就要用到 Linux 内核的 Cgroups 技术了。

Linux Cgroups

Linux Cgroups（Control Groups）主要负责对指定的一组进程做资源限制，同时可以统计其资源使用。具体包括 CPU、内存、存储、I/O、网络等资源。

我们有了 Cgroups，就不用担心某一组□容器进程突然将计算机的全部物理资源占满这种问题了，可以方便地限制和实时地监控某一组容器进程的资源占用。

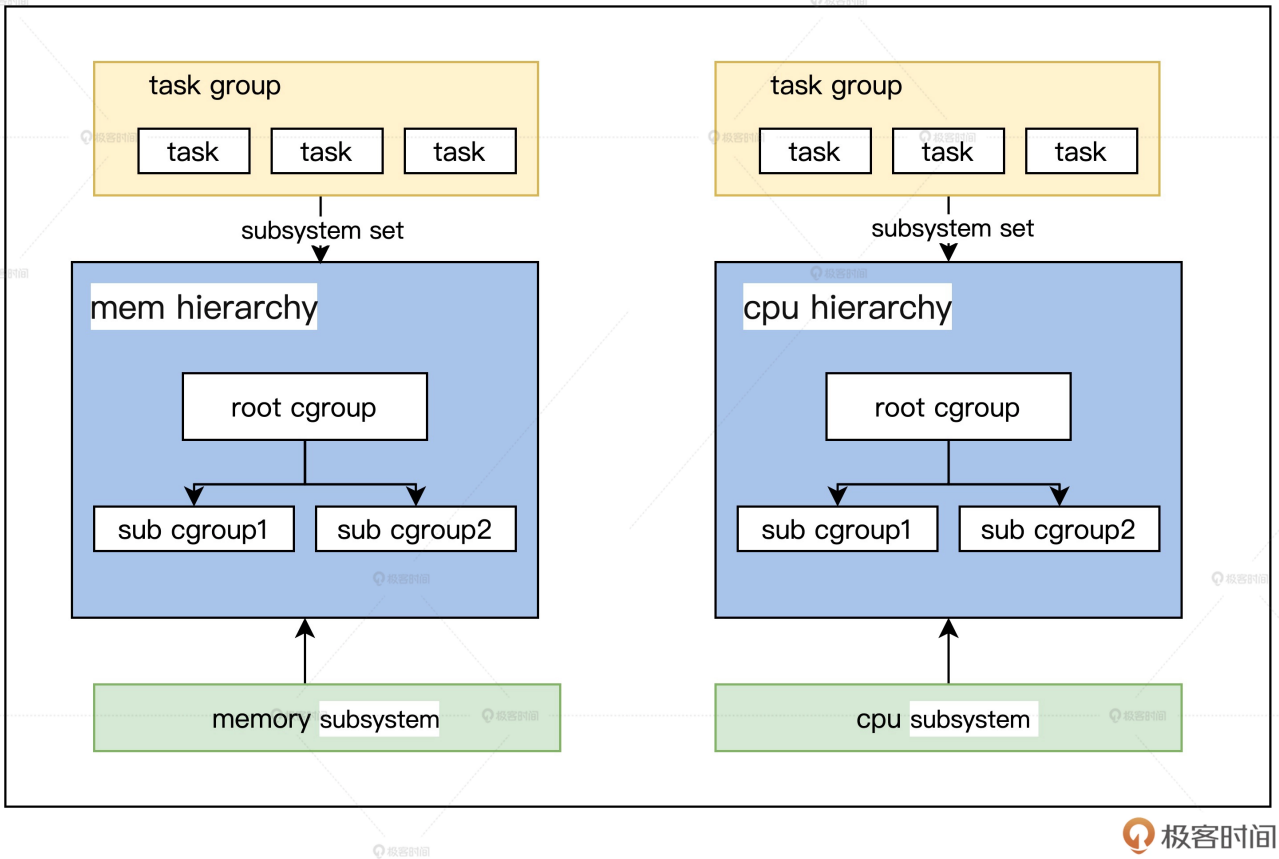
Cgrpups 包含几个核心概念，分别是 **Task (任务)、Control Groups（控制组 ）、subsystem（子系统）、hierarchy（层级数）**。

Task: 任务，在 Cgroup 中，任务同样是一个进程。

Control Groups：控制组，Cgroups 的一组进程，并可以在这个 Cgroups 通过参数，将一组进程和一组 linux subsystem 关联起来。

subsystem：子系统，是一组资源控制模块，subsystem 作用于 hierarchy 的 Cgroup 节点，并控制节点中进程的资源占用。

hierarchy：层级树 Cgroups，将 Cgroup 通过树状结构串起来，通过虚拟文件系统的方式暴露给用户。



cgroup示意图

Linux 内核提供了很多 Cgroup **subsystem** 参数，我们了解一下容器中常用的几类。：

subsystem参数	作用
cpu group	控制一组容器进程最大的CPU使用量
memory group	控制容器中最大的内存使用量
device group	控制在容器中看到的device设备，保障安全
freezer group	容器停止时Freezer当前容器进程都写入cgroup、进程冻结
blkio group	容器停止时Freezer当前容器进程都写入cgroup、进程冻结
blkio group	限制容器到磁盘的IOS、BPS
pid group	限制容器里面可用到的最大进程数量

极客时间

Linux 内核提供了很多 Cgroup 驱动，容器中常用的是下面两种。

1.Cgroupfs 驱动：需要限制 CPU 或内存使用时，直接把容器进程的 PID 写入相应的 CPU 或内存的 cgroup。

2.systemdcgroup 驱动：提供 cgroup 管理，所有的 cgroup 写操作需要通过 systemd 的接口来完成，不能手动修改。

了解了 Cgroups **subsystem** 的类型，那么容器到底要怎么调用内核才能配置 Cgroups 呢？我们动手实验下，才会有更深的体会。

新建 Cgroup 挂载文件

首先我们试下新建一个 Cgroup，名为 cgroup-cosmos，我们先创建一个 hierarchy，再进行挂载代码如下。

复制代码

```
mkdir cgroup-cosmos
sudo mount -t cgroup -o none,name=cgroup-cosmos cgroup-cosmos cgroup-cosmos/
ll ./cgroup-cosmos
```

```
cgroup.clone_children  cgroup.procs  cgroup.sane_behavior  notify_on_release
[root@beautiful-poke-2 cgroup-test]# ll ./cgroup-cosmos
total 0
-rw-r--r-- 1 root root 0 Aug  3 00:46 cgroup.clone_children
-rw-r--r-- 1 root root 0 Aug  3 00:46 cgroup.procs
-r--r--r-- 1 root root 0 Aug  3 00:46 cgroup.sane_behavior
-rw-r--r-- 1 root root 0 Aug  3 00:46 notify_on_release
-rw-r--r-- 1 root root 0 Aug  3 00:46 release_agent
-rw-r--r-- 1 root root 0 Aug  3 00:46 tasks
[root@beautiful-poke-2 cgroup-test]#
```

可以看到我们生成了 Cgroup 的几个配置文件，这些就是 hierarchy 根节点的配置文件。

创建子 Cgroup

接下来，我们要创建子 Cgroup，名为 cgroup-cosmos-a。

复制代码

```
cd cgroup-cosmos
```

```
mkdir cgroup-cosmos-a
tree
```

可以看到，我们在根节点下新建一个目录，会默认识别为一个子 Cgroup，而且它会继承父级的配置。

```
[root@beautiful-poke-2 cgroup-cosmos]# tree
.
|-- cgroup.clone_children
|-- cgroup-cosmos-a
|   |-- cgroup.clone_children
|   |-- cgroup.procs
|   |-- notify_on_release
|   `-- tasks
|-- cgroup.procs
|-- cgroup.sane_behavior
|-- notify_on_release
|-- release_agent
`-- tasks
```

在 Cgroup 中添加、移动进程

目录建好了，添加、移动进程的操作也很简单，我们只要将当前的进程 ID 写入对应的 cgroup 文件即可，代码如下。

复制代码

```
echo $$ // 583
cat /proc/583/cgroup
```

```
[root@beautiful-poke-2 cgroup-cosmos]# echo $$
583
[root@beautiful-poke-2 cgroup-cosmos]# cat /proc/583/cgroup
13:name=cgroup-cosmos:/
12:name=cgroup-test:/
11:devices:/
10:hugetlb:/
9:memory:/
8:perf_event:/
7:freezer:/
6:cpuset:/
5:pids:/
4:blkio:/
3:cpu,cpuacct:/
2:net_cls,net_prio:/
1:name=systemd:/user.slice/user-0.slice/session-1388.scope
```

结合图里的代码我们发现，当前的 583 进程是在 cgroup-cosmos 下，现在我们将终端进程移动到 cgroup-cosmos-a。

复制代码

```
cd cgroup-cosmos-a
sh -c "echo $$ >> tasks"
```

```
[root@beautiful-poke-2 cgroup-cosmos-a]# cat /proc/583/cgroup
13:name=cgroup-cosmos:/cgroup-cosmos-a
12:name=cgroup-test:/
11:devices:/
10:hugetlb:/
9:memory:/
8:perf_event:/
7:freezer:/
6:cpuset:/
5:pids:/
4:blkio:/
3:cpu,cpuacct:/
2:net_cls,net_prio:/
1:name=systemd:/user.slice/user-0.slice/session-1388.scope
```

可以看到。现在 583 进程已经移动到 group-cosmos: /group-cosmos-a 目录下了。

限制 Cgroup 中进程的资源

前面我们曾经说过 Cgroup 可以限制资源，那具体要怎么操作呢？

前面我们创建的 hierarchy 其实并没有关联到任何的 subsystem，所以没办法通过它来限制资源使用。但是系统给每个 hierarchy 都制定了默认的 subsystem，我们看一下具体代码。

复制代码

```
# 查看hierarchy的subsystem，为/sys/fs/cgroup/memory/
mount | grep memory
cd /sys/fs/cgroup/memory/
sudo mkdir cosmos-limit-memory
cd cosmos-limit-memory
ls
```

```
[root@beautiful-poke-2 memory]# cd cosmos-limit-memory
[root@beautiful-poke-2 cosmos-limit-memory]# ls
cgroup.clone_children           memory.kmem.tcp.max_usage_in_bytes  memory.oom_control
cgroup.event_control            memory.kmem.tcp.usage_in_bytes      memory.pressure_level
cgroup.procs                    memory.kmem.usage_in_bytes          memory.soft_limit_in_bytes
memory.failcnt                  memory.limit_in_bytes               memory.stat
memory.force_empty              memory.max_usage_in_bytes           memory.swappiness
memory.kmem.failcnt             memory.memsw.failcnt                memory.usage_in_bytes
memory.kmem.limit_in_bytes      memory.memsw.limit_in_bytes         memory.use_hierarchy
memory.kmem.max_usage_in_bytes  memory.memsw.max_usage_in_bytes     notify_on_release
memory.kmem.slabinfo            memory.memsw.usage_in_bytes         tasks
memory.kmem.tcp.failcnt         memory.move_charge_at_immigrate
memory.kmem.tcp.limit_in_bytes  memory.numa_stat
```

先启动一个未限制的进程，代码如下。

复制代码

```
stress --vm-bytes 200m --vm-keep -m 1
```

代码运行后的结果如下图所示。

```
Complete!
[root@beautiful-poke-2 cosmos-limit-memory]# stress --vm-bytes 200m --vm-keep -m 1
stress: info: [3128] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
```

现在我们设置最大内存，并且将进程移动到当前 Cgroup 中，在此运行一个进程。这样操作以后，我们就可以通过 top 命令看到已经将 stress 的最大内存限制到 100m 了。

复制代码

```
# 设置最大内存占用
sh -c "echo "100m" > memory.limit_in_bytes"
# 移动到这个cgroup内
sh -c "echo $$ >> tasks"
# 再启动一个机型
stress --vm-bytes 200m --vm-keep -m 1
top
```

好，说到这儿相信你已经对 **Namespace** 和 **Cgroups** 这两大技术建立了初步认知，它们是 Linux 操作系统下开发容器的最基本的技术。

总结与思考

好，这节课的内容告一段落了，我来给你做个总结。

首先我们了解了到底什么是容器，以 Docker 为蓝本分析了容器的基础功能架构，包括客户端（**Client**）、管理进程（**Host**）、镜像仓库（**Registry**）三大部分。

引擎进程（Host）是 Docker 的核心，包括引擎进程（**Daemon**）、驱动（**Driver**）、容器管理包（**Libcontainer**）、镜像（**Images**）。

用户通过 Client 与 Daemon 建立通信，并发送请求给后者；而 Daemon 作为 Docker 架构中的核心部分，其中的 Server 负责接收 Client 发送的请求，而后 Engine 执行 Docker 内部的一系列工作，每一项工作都是以一个 Job 的形式的存在；在执行 Job 的过程中，我们会使用下层的 Driver（驱动）来完成工作，driver 通过 libcontainer 来访问内核中与容器相关的 API，从而实现具体对容器进行的操作。

之后我们分析了一个容器如何通过各种内核提供的技术（**NameSpace，Cgroup，UnionFS** 等技术）的组合运行起来，提供对外访问隔离功能。

其实容器的技术本身没有太大的技术难度，容器就本质上就是一种**特殊的进程**，利用了操作系统本身的资源限制和隔离能力，通过约束和修改进程的动态表现，从而为其创造出一

个“边界”——也就是独立的"运行环境"，有兴趣的同学可以深入了解 Docker 的源码，并可以自己尝试重新实现一个简单的的容器。

思考题

在我们启动容器后，一旦容器退出，容器可写层的所有内容都会被删除。那么，如果用户需要持久化容器里的部分数据该怎么办呢？

欢迎你在留言区跟我交流，也欢迎你把这节课分享给朋友。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 43 | 虚拟机内核：KVM是什么？

下一篇 用户故事 | 成为面向“知识库”的工程师

更多课程推荐

精选留言 (2)

写留言

pedro 酥
2021-08-18

持久化使用 Volume。

Volume 的原理其实也简单：容器的根目录需要挂载到宿主机下，然后通过 chroot 改变容器根目录，因此对于 Volume 文件，只需将其挂载到宿主机上对应的容器根目录中，而后容器内部引起的文件变更，也会改变 Volume 文件。

展开

作者回复: 哈哈

4

Fan 酥
2021-08-18

在我们启动容器后，一旦容器退出，容器可写层的所有内容都会被删除。那么，如果用户需要持久化容器里的部分数据该怎么办呢？

映射到宿主机上。

展开

作者回复: 是的